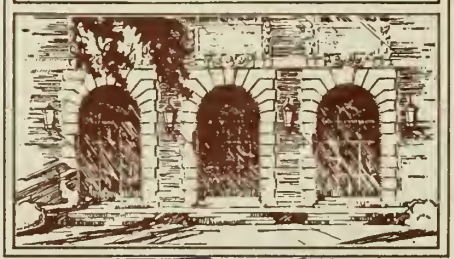
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 800 - 805
cop. 2



Digitized by the Internet Archive
in 2013

http://archive.org/details/universalcrossas803komi

UIUCDCS-R-76-803

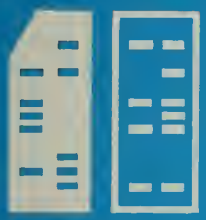
510.84
Ill6r
no. 803
cop 2

A UNIVERSAL CROSS-ASSEMBLER

By

CHARLES PATRICK KOMINCZAK

May, 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-76-803

A UNIVERSAL CROSS-ASSEMBLER

BY

CHARLES PATRICK KOMINCZAK

May 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the Department of Computer Science and the Department of Electrical Engineering, and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering, 1976.

ACKNOWLEDGMENTS

I would like to thank Dr. Michael Faiman for his support and many helpful suggestions during the preparation of this manuscript. I am especially indebted to Scott Davidson for his aid in procuring the original version of the program. Finally, thanks go to all of my colleagues who indulged me in conversation on assemblers in general, and thereby influenced the development of the program.

TABLE OF CONTENTS

		Page
I.	INTRODUCTION	1
II.	FUNCTIONAL SPECIFICATION	7
	Source Languages	7
	Pseudo-ops	12
	Correspondence Between Source Language and Hardware	12
III.	USE OF THE ASSEMBLER	19
	Running Programs Through the Assembler	19
	Implementation of New Source Languages	21
	Criticism of the Program	27
IV.	APPLICATIONS	33
	Microprogramming	33
	Universal Assembly Language	34
V.	SUGGESTIONS FOR FUTURE WORK	37
	Conversational Assembler	37
	Macro Assembler	38
	Relocatable Output	39
	Conclusion	39
REFERENCES		41
APPENDIX		42
	Standard Pseudo-ops	43
	Description of EBF Notation	44

I. INTRODUCTION

In the software development cycle for any new computer system, the first item implemented is usually a cross-assembler. Using a high level language (HLL), this typically requires from four to eight hundred programmer hours [1], while an equivalent version coded in assembly language (AL) will increase implementation time by fifty per cent. Thus, it is clear that software related expenditures will be the primary component of the cost of most microcomputer systems. Hence, the primary motivation of the MUMS (Modular Universal Microcomputer System) assembler project is to reduce the gap between the hardware and software investments.

In the past, cross-assemblers were sometimes considered to be short-term software. After a significant amount of software was developed on some host system, activity would gradually shift over to the target machine. In contrast, present cross-assemblers are very likely to remain important software development tools for microprocessors, mainly because the microcomputer is such a hostile environment for a resident assembler. These machines are typically configured with small memories, have limited instruction

sets, and use very slow memory or I/O devices. Hence, it is usually more convenient to program on a powerful host system with time-sharing facilities, especially when a large number of programmers are involved.

For the same reasons, cross-compilers have been gaining in popularity. However, it has been observed that most compilers seldom generate code that is within ten percent in length of the same program written in AL [2]. In fact, most compilers usually do much worse. For a given microcomputer system, a programmer is often forced to squeeze his program into some specified amount of ROM. Hence, the inefficient code generated by a compiler may not be tolerable. If the compiler has the option of translating the source into AL mnemonics, the programmer may be able to recode at this lower level to increase efficiency. Otherwise, he is probably better off coding the entire program in AL. Other situations in which AL programming is preferable are discussed in [3]. Thus, it appears that cross-assemblers are going to be useful for much of the foreseeable future.

Initially, the MUMS assembler was conceived as a sort of "instant assembler." Upon execution, the program would initialize internal tables and switches according to the contents of an external file. This file would consist of all machine dependent information, e.g., mnemonics and corresponding object codes, default values for certain assembly functions, and a set of "templates." As used here,

the word template may be interpreted as a mask used for building the machine code equivalent of a mnemonic statement. Thus, any programmer with an understanding of basic assembly techniques could bring up a cross-assembler very quickly by simply constructing the proper external file for the target machine.

The assembler itself would contain all of the functions thought central to the assembly process, e.g., line scanning, table look-up, etc. A search of the opcode table would return a basic opcode bit pattern and a template number. A template look-up would then supply the details for processing the remaining fields of the instruction. The templates, as specified in the external file, could be visualized as a set of keywords with optional arguments. During execution, the assembler would translate these keywords into procedure calls. Hence, the operand processor for the assembler would be implemented as a collection of independent procedures, with the number of operands to be processed and with the particular procedures to be employed determined by the sequence of keywords in the template.

These concepts characterize an ideal universal assembler. To handle new assembly languages, the user should be concerned only with the specifics of the machine file, and not with the implementation details of the assembler itself. In practice, however, the templates will require a defining language which is almost as complex as

some high level languages. Such extreme generality is necessary in order to handle the wide range of special cases normally encountered in the implementation of assemblers. If one simply chooses some fixed set of source operations, then he is faced with the difficult task of constructing a logical proof that the set is sufficient to handle all special cases, or he must be prepared to add fixes whenever these special cases appear. Thus, increased generality increases the complexity of the assembler. In effect, the assembler becomes a template compiler and pays a severe time penalty during every execution. From the viewpoint of software maintenence, it is desirable for the assembler to remain as simple as possible. If the assembler is to be used often, a premium must also be placed on efficiency.

The compromise to be proposed here is basically an assembler skeleton. This skeleton performs all of the bookkeeping chores, e.g., table management, line scanning and listing, and paper tape generation. Thus, a custom cross-assembler may be implemented by simply inserting the template definitions into the skeleton and by creating a new external opcode table. For maximum execution efficiency, the skeleton should obviously be written in AL. However, ease of maintenance and overall flexibility were deemed more important, so that a HLL therefore seemed more appropriate.

This scheme offers all of the advantages of HLL programming and is more efficient than storing the templates externally. In addition, the technique is so flexible that it will usually be possible to mimic the manufacturer's AL specifications. Existing software can then be modified very quickly to pass through the new assembler. Using this approach, a programmer should be able to bring up a new cross-assembler in less than one week. These ideas have been tested by implementing a single assembler which generates code for the National IMP-16, Intel 8080, RCA COSMAC, Fairchild F8, and the MOS Technology 650X.

The technique described above is similar to that discussed by Conley [4]. Conley suggests the use of BASIC as the implementation language because of its wide availability, and presents a fair critique of its use in this application. The MUMS assembler was written in PASCAL, and is an extension of a program written by Jones and Lathrop [5]. This early work provided a quick start for the project, and so the decision was made to continue with PASCAL. This decision is difficult to justify in terms of portability, especially since DEC-10 PASCAL imposes several non-standard conventions. However, the use of PASCAL is usually encouraged on the grounds that, since it is procedure-oriented, the resulting code is self-documenting and highly modular.

Earlier work in this area was reported in the late sixties ([6] and [7]). Georgiou's "Generalized Assembler" is coded in IBM 360 AL, and therefore conflicts with the philosophies detailed above. It also lacks the important element of convenience, as the local 360 does not directly support time-sharing. In addition, both programs ([6] and [7]) were designed specifically for a certain class of machines, namely the PDP-7, -8, and -9, computers. Although reference [7] was not available at the time of this writing, from available evidence it seems that both programs are closely tailored to match their target machines, and therefore lack the flexibility to adapt easily to current microprocessors.

II. FUNCTIONAL SPECIFICATION OF THE ASSEMBLER

An underlying assumption of the MUMS assembler is that the significant differences between assembly languages occur only in the operand fields of each instruction. Hence, the syntactic and semantic processing of each operand for a particular opcode may be accomplished by proper coding of the template. Conceptually, the assembler can be partitioned into two parts. One part handles most of the usual functions of an assembler. The second part is the set of templates, which must be customized for a given machine. The remainder of this section will attempt to define the responsibilities of each part and to show that this organization is sufficient to handle most machines.

Source Languages

The assembler processes mnemonic source statements in a free field format. Tables 1 and 2 present an EBF description of legal source statements. In order to keep the notation simple, the lexical description is somewhat informal. Hence, the following facts are included to supplement the lexical details of Table 2. An explanation of the EBF notation appears in the appendix.

TABLE 1

EBF Description of Source File

```
<PROG>   ::=  <STMT>+  <END>    EOF
<STMT>   ::=  (ID COLON)*
              (  ((OPCODE | PSEUDO-OP)
                    (<OPERAND>++<DELIM>)?)
               | ((ID | DOT) EQ-OP <OPERAND>)  )?
               (SEMI COMMENT)?
<DELIM>  ::=  COMMA | SEMI | LP | RP | CR-LF
<OPERAND>::=  (DOT | ID | NUM)(<OPERATOR><OPERAND>)?
         ::=  LAB <OPERAND> RAB
         ::=  <UNARY-OP><OPERAND>
<OPERATOR>::= PLUS-OP | MINUS-OP | AND-OP | OR-OP
              | MUL-OP | DIV-OP
<UNARY-OP>::= PLUS-OP | MINUS-OP | COMP-OP
<END>    ::=  END-PSEUDO (<OPERAND>)?
```

TABLE 2

Textual Representation of Terminal Symbols

```
LP        (                 RP          )
COLON     :                 PLUS-OP     +
SEMI      ;                 MINUS-OP    -
COMMA     ,                 AND-OP      &
DOT       .                 OR-OP       !
SQUOTE    '                 DIV-OP      /
DQUOTE    "                 MUL-OP      *
LAB       <                 COMP-OP     ¬
RAB       >                 EQ-OP       =

PSEUDO-OP      .ASCII | .BYTE | .FORMAT | .LIST
             | .NLIST | .NPUNCH | .PAGE | .RADIX
             | .TABLE | .WORD

OPCODE      ID specified by external opcode table.


    <alph>  ::=  'A' | ... | 'Z'

    <dig>   ::=  '0' | ... | '9'

<alph-dig>  ::=  'A' | ... | 'F'

    <id>    ::=  (DOT)?<alph>(<alph>|<dig>|DOT)*  --> ID

    <num>   ::=  ('|' <radix>)?

                     <dig>(<dig>|<alph-dig>)     --> NUM

            ::=  SQUOTE <any>                    -->

            ::=  DQUOTE <any><any>               -->

    <radix> ::=  'B' | 'O' | 'D' | 'H'

    <any>   ::=  (any printing ASCII character,

                    except underscore)

<comment>   ::=  <any>*                      -->  COMMENT
```

1. Source statements are contained in a single line with a maximum length of eighty columns. A line is terminated by a carriage-return/line-feed (CR-LF).

2. A terminal symbol can be bounded by an arbitrary number of spaces.

3. The non-terminal <OPERAND> actually specifies the class of expressions which may be evaluated by the expression analyzer. Precedence is enforced by the use of angled brackets (less-than and greater-than), thereby freeing parentheses for special applications. The particular delimiter (<DELTM>) used to separate two expressions is a semantic issue, and must therefore be resolved in the template. An occurrence of DOT in an expression is a reference to the current value of the location counter.

Obviously, certain constructs generated by these rules are impractical. For example, long labels waste table space, while large numbers may cause a truncation error. Also, numbers are evaluated in the context of the current radix. Hence, for radix eight, the digit nine would cause an error message to be printed. The current radix may be temporarily overridden by use of up-arrow(radix), where the temporary radix is B for binary, O for octal, D for decimal, and H for hex.

The phrase structure grammar of Table 1 concisely describes the syntactic aspects of programs acceptable to the assembler. A program is simply a collection of one or more statements terminated by the .END pseudo-op. Statements may consist of three parts: any number (zero or more) of labels, followed by an optional opcode, pseudo-op, or symbol definition part, followed by an optional comment part. Note that blank lines are legal. Note also that the only time a user defined template is executed is when an opcode appears in the statement. The template specifies how the operands, if present, should be processed. In most cases, the template will simply call the expression analyzer once for each operand and use the results to construct the object code. However, it is possible to write template programs that search for arbitrary tokens in the operand field. In effect, then, the user has the ability to alter the grammar and assign appropriate semantics to the result. For example, this facility may be used to distinguish between the addressing modes of an instruction, or to scan for predefined register mnemonics. It is this great flexibility in template design that allows the assembler to be easily adapted to a wide variety of assembly languages.

Pseudo-ops

The pseudo-ops listed in Table 2 are typical of those found in most assemblers. They are evaluated by a fixed set of templates, and any subset may be implemented for a given machine. No claim is made as to their universality. Should this set prove inconvenient, the user may easily install new pseudo-op templates to suit his needs. A brief description of each of the supplied pseudo-ops appears in the appendix.

Correspondence Between Source Language and Hardware

The manner in which certain instructions execute on a given machine often places subtle requirements on the construction of the object code. Basically, most of these problems are concerned with proper incrementation of the location counter. Some of the problems place stringent requirements on the design of the assembler skeleton, but many can be handled at the template level. The following special cases are not exhaustive, since an exhaustive study would necessarily have to include all machines. Rather, these cases are meant to be indicative of the types of problems that can occur during the design of an assembler. Any problems overlooked here will hopefully prove easy to solve, given the flexibility of the program and the power of PASCAL.

One of the more inconvenient problems encountered in designing a general purpose assembler is the fact that many machines have instructions which are variable in length. In these cases, the length of the instruction must be determined by an examination of the operands. For example, the MCS650X has a "zero page" addressing mode utilizing a one byte address, in addition to the more general form having a two byte address. Thus, an instruction such as

ORA X

may assemble into two or three bytes depending on whether X is assigned to page zero or not. Since programmers generally prefer to order their code in the same order as it will appear in memory, it seems reasonable to assume that page zero labels and source code will appear near the beginning of the program. With a zero page addressing mode, it is usually advantageous to keep frequently referenced data on page zero. Thus, most memory references appearing later in the program will generally be to page zero or to nearby locations (principle of locality of reference). The assembler should therefore default to absolute mode addressing when the operand is indeterminate and the opcode permits zero page reference. This requires the user to be careful that forward references to page zero labels do not occur.

Traditionally, the first pass of an assembler has been considered responsible only for construction of the symbol table. Hence, the first pass was usually designed to run as

quickly as possible. This is practical only if the length of each instruction is known in advance. To determine this length, however, it is clear that the assembler must attempt to evaluate the source statement completely; otherwise the location counter will not be properly maintained, and stored labels may be assigned incorrect values. For simplicity, the assembler therefore assembles the source file in two identical passes. An internal switch prevents any output during pass one, thereby suppressing spurious error messages.

Unfortunately, this simple approach involves a duplication of effort, since the table searching for labels and opcodes is repeated in the second pass. Many assemblers get around this problem by producing a temporary file during pass one. Pass two then needs only to scan the intermediate code and reprocess the operand field in order to construct the object code. Besides simplicity, however, the simpler approach offers an interesting bonus. If the location counter becomes incorrect because of an indeterminate operand during pass one, succeeding labels will have incorrect symbol table entries. During pass two, these labels will be flagged as multiply defined. This type of error would otherwise go undetected.

One should also be aware of the differences involved in computation of branch addresses. During execution, many machines increment their program counter before adding

branch offsets. The assembler should then compute the offset from the address following the instruction. However, for the F8, the offset is from the first byte of the branch instruction. This detail is easily resolved in the template.

The final problem involving location counter maintenance requires a distinction to be made between machines that are byte-addressable and machines that are word-addressable. For the word-addressable machine, the location counter is simply incremented once for each word of object code produced, whereas the byte-addressable machine must add N (bytes per word). In some cases, it may also be appropriate to check for word boundary errors. For example, in the PDP-11, a 16-bit word may not start at an odd address. As before, putting the location counter under template control permits an easy solution of these problems.

The templates are contained in a single PASCAL procedure. Besides contributing to software modularity, this structure provides a simple solution to the assembly of certain awkward opcodes. As an example, consider the various mnemonic operand combinations for the LR instruction of the Fairchild F8. This instruction assembles into the seemingly random bit patterns listed in Table 3. The number of cases involved suggests that a rather large and messy template would be required. However, the problem has a simple recursive solution. First, concatenate the opcode

and operand fields to form a new "opcode," and then execute the template associated with this new "opcode." This second template, which is invoked by a recursive call of the template procedure, will trivially push the stored "opcode" bit pattern into the object buffer. This technique permits a direct means of translating mnemonic phrases into arbitrary bit patterns and results in very simple templates. On the negative side, however, this technique results in a much larger opcode table for this example. Searching this larger table increases the assembly time for all instructions.

TABLE 3

LR instruction of F8

Mnemonic		Machine Code
LR	DC,Q	0F
LR	DC,H	10
LR	PC,Q	0D
LR	P,K	09
LR	Q,DC	0E
LR	H,DC	11
LR	K,P	08
LR	A,r	4r
LR	A,KU	00
LR	A,KL	01
LR	A,QU	02
LR	A,QL	03
LR	r,A	5r
LR	KU,A	04
LR	KL,A	05
LR	QU,A	06
LR	QL,A	07
LR	IS,A	0B
LR	W,J	1D
LR	A,I	0A
LR	J,W	1E

The preceding discussion was given to help justify the final form of the assembler. Georgiou's "generalized assembler" is, in comparison, a very rigidly specified software package. The additions to his program which specify a new machine really only change insignificant formatting details (his program reads fixed field input). The program therefore translates a fixed grammar according to a fixed set of semantics. Thus, the program is able to assemble code only for the class of machines for which it is designed. Furthermore, there is nothing in Georgiou's thesis to suggest that it would be simple to modify his work to handle current microprocessors. In contrast, the MUMS assembler surmounts these problems through use of a flexible assembler skeleton and user defined templates. The use of HLL allows rapid template design and debugging, but also results in a relatively less efficient program. In terms of convenience and maintainability, however, the program is felt to be an excellent compromise.

III. USE OF THE ASSEMBLER

Now that some justification has been given for the structural design of the assembler, it seems appropriate to relate some details concerning its use. The current version of the assembler contains extensions which facilitate its use in a time-shared environment. The host computer is a DEC PDP-10, and the assembler therefore shares the flavor of other DEC software. The virtues of time-sharing are well-known, and hence need not be elaborated here. The scope of the following material will be limited to the procedural aspects of the program. Further information on the implementation details is available in two forms - a user's manual and a maintenance manual.

Running Programs Through the Assembler

After creating or editing a source file with one of the system text editors, this file may be assembled by issuing the following monitor command.

```
RUN ASSEM [4577,5240]  14
```

The number in brackets is a project-programmer number, which specifies the disk area where the assembler currently resides. The remaining number indicates the amount of core needed to run the program. At the start of execution, ASSEM

will prompt with an asterisk. The user is then expected to enter a filename with the following form.

(<alph> ((<alph>|<dig>)*+5)? DOT EXTENSION)

Currently valid extensions are listed in Table 4. If, for example, the source file was named PROG.INT, the assembler would

1) load the internal opcode table from file OP.INT,

2) open file PROG.LST for an INTEL 8080 assembly listing, and

3) if the .NPUNCH directive is not detected during pass one, open file PROG.TAP for the absolute object code generated by pass two. Source lines in error and error messages are then printed at the terminal. The program will return control to the monitor after printing the number of errors. The user should then correct the source and iterate as necessary.

TABLE 4

Valid Source Extensions

EXTENSION	MACHINE
INT	INTEL 8080
MOS	MOS TECHNOLOGY 650X
F8	FAIRCHILD F8
COS	RCA COSMAC
IMP	NATIONAL IMP-16

Implementation of New Source Languages

In the previous section, it was explained that the assembler selects the opcode file at run-time according to the type (extension) of source file entered. Hence, one may implement a new language simply by creating a new opcode file OP.(unique extension) and by installing new templates in the skeleton. As an example, these additions will be considered for the IMP-16. This machine was chosen because its source code is reasonably complex yet straightforward to assemble. In particular, this example should demonstrate the assertion made in the introduction - that storing templates externally for interpretive execution adds needless complexity to the assembler.

The IMP-16 assembler is implemented by the additions of Figures 1 and 2(pp. 25-27). The first line of the opcode file is a title which is printed on each page of the assembly listing. The second line sets internal switches in the assembler. 'H' sets a default hex listing mode for addresses and object code. The next digit specifies the number of hex digits needed to encode the address space. An optional character not appearing in this case is 'B', which indicates a byte-addressable machine. The default is word-addressable. This switch is used internally to select between byte and word options. For example, a full word checksum will be generated for an IMP-16 paper tape load block.

```
NATIONAL IMP-16 ASSEMBLER
H4
.ASCII          6   0
.END           01   0
.FORMAT        12   0
.LIST          07   0
.NLIST         10   0
.NPUNCH        13   0
.PAGE          11   0
.RADIX         05   0
.TABLE         14   0
.WORD          03   0
ADD        140000  23
AISZ       044000  29
AND        060000  24
BOC        010000  27
CAI        050000  29
DSZ        076000  25
HALT       000000  33
ISZ        074000  25
JMP        020000  26
JSR        024000  26
JSRI       001600  28
LD         100000  22
LI         046000  29
OR         064000  24
PFLG       004200  34
PULL       042000  31
PULLF      001200  33
PUSH       040000  31
PUSHF      000200  33
RADD       030000  32
RAND       030203  32
RCPY       030201  32
RIN        002000  28
ROL        054000  29
ROR        054000  30
ROUT       003000  28
RTI        000400  28
RTS        001000  28
RXCH       030200  32
RXOR       030202  32
SFLG       004000  34
SHL        056000  29
SHR        056000  30
SKAZ       070000  24
SKG        160000  23
SKNE       170000  23
ST         120000  22
SUB        150000  23
XCHRS      052000  31
```

Figure 1 IMP-16 opcode file

The remaining lines of the opcode file are stored in the opcode table. Each record in this table is composed of three fields: a basic opcode bit pattern (second column of Figure 1, in octal), a template number (third column, in decimal), and a pointer into a character array where the textual representation of the opcode is stored. Note that all of the pseudo-ops share the same template number. In Figure 2, template zero calls the pseudo-op evaluator, which in turn executes the pseudo-op template selected by a case on the stored "opcode" bit pattern. Similarly, normal opcodes case directly to the appropriate template. If an opcode is not found in the table, the OPSRCH function returns a pointer to an error record. This results in the selection of reserved pseudo-op template number zero, thereby printing an error message.

The assembler skeleton provides several procedures and functions which simplify the task of template coding. EXPRESSION evaluates the non-terminal <OPERAND> according to the productions of Table 1. ERROR globalizes error handling, while OUTWORD automates insertions into the object buffer. In addition, two user defined functions provide further template simplifications. With minor variations, a large group of IMP instructions have the following form [9].

OPCODE r,@disp(xr)

Function EA evaluates the effective address given by the second operand field. All three parts of this field are treated as optional. If indirect addressing is legal for

the given instruction, then IND (not equal to zero) will specify the location of the indirect bit. This bit is then set if '@' is present in the source. Note that the default addressing mode is relative to the program counter (XR=1). The second function, DISP, truncates the displacement to eight bits and prints one of two error messages if the truncated bits are not all zeroes or all ones.

The preceding discussion indicates the relative ease with which new syntax rules and semantics can be coded into templates. Error checking can be as elaborate as desired. For example, the registers accessible to the programmer are AC0, AC1, AC2, and AC3. In template twenty-four, however, only AC0 and AC1 are legal. Of course, if a programmer wishes to use these register mnemonics, he must declare them in symbol definitions. Alternatively, a procedure could have been written to recognize these mnemonics. Hence, this implementation is by no means a unique solution. The implementor is free to include as many features that suit him.

```
PROCEDURE PROCESSOPCODE;
VAR TAND1,TAND2: INTEGER;
FUNCTION DISP(ERR: ERRORTYPE): INTEGER;
BEGIN
     DISP:=0;
     IF (ANDF(TAND2,HFF00)<>0)&(ANDF(TAND2,HFF00)<>HFF00)
     THEN ERROR(ERR)
     ELSE DISP:=ANDF(TAND2,H00FF)
END;
FUNCTION EA(IND: INTEGER): INTEGER;
/*  DECODES ADDRESSING MODES FOR THE IMP-16 */
VAR  XR: INTEGER;
BEGIN
     XR:=1;
     TAND2:=0;
     IF ¬EOLF
     THEN BEGIN
               FIRSTCH(CH);
               IF (CH='@')
               THEN IF (IND<>0)
                    THEN TAND1:=TAND1+IND
                    ELSE ERROR(BADEX)
               ELSE LINBUFP:=PRED(LINBUFP);
               IF LINBUF[SUCC(LINBUFP)]='('
               THEN LINBUFP:=SUCC(LINBUFP)
               ELSE TAND2:=EXPRESSION;
               IF ¬EOLF THEN XR:=EXPRESSION;
               IF (XR=1)
               THEN TAND2:=TAND2-SUCC(LLOCC);
          END;
     EA:=(4*TAND1+XR)*256 + DISP(BBRAN);
END;

BEGIN
          OPID:=OPSRCH(OPLEN);
          OPCODE:=OP[OPID].CODE;
          /*DECODE AND PROCESS OPCODE*/
          CASE OP[OPID].MODE OF
               0: PSEUDOOP;
        /* TEMPLATES 22:34 PROCESS IMP-16 OPCODES */
               22: BEGIN       /* LD,ST */
                        TAND1:=EXPRESSION;
                        IF TAND1>3 THEN ERROR(NOREG);
                        OUTWORD(EA(4)+OPCODE)
                    END;
               23: BEGIN       /* ADD,SUB,SKNE,SKG */
                        TAND1:=EXPRESSION;
                        IF TAND1>3 THEN ERROR(NOREG);
                        OUTWORD(EA(0)+OPCODE)
                    END;
```

Figure 2 IMP-16 opcode templates

```
24: BEGIN       /* AND,OR,SKAZ */
        TAND1:=EXPRESSION;
        IF TAND1>1 THEN ERROR(NOREG);
        OUTWORD(EA(0)+OPCODE)
    END;
25: BEGIN       /* DSZ,ISZ */
        TAND1:=0;
        OUTWORD(EA(0)+OPCODE);
    END;
26: BEGIN       /* JMP,JSR */
        TAND1:=0;
        OUTWORD(EA(1)+OPCODE)
    END;
27: BEGIN       /* BOC */
        TAND1:=EXPRESSION;
        TAND2:=EXPRESSION-SUCC(LLOCC);
        IF (TAND1>15) THEN ERROR(BADEX);
        OUTWORD(OPCODE+256*TAND1
                        +DISP(BBRAN));
    END;
28: BEGIN       /* JSRI,RIN,ROUT,RTI,RTS */
        IF ¬EOLF
        THEN TAND1:=ANDF(EXPRESSION,H007F)
        ELSE TAND1:=0;
        OUTWORD(OPCODE+TAND1)
    END;
29: BEGIN       /* AISZ,CAI,LI,ROL,SHL */
        TAND1:=EXPRESSION;
        IF (TAND1>3) THEN ERROR(NOREG);
        IF ¬EOLF
        THEN TAND2:=EXPRESSION
        ELSE TAND2:=0;
        OUTWORD(OPCODE+256*TAND1
                        +DISP(BTRNC))
    END;
30: BEGIN       /* ROR,SHR */
        TAND1:=EXPRESSION;
        IF (TAND1>3) THEN ERROR(NOREG);
        TAND2:=-EXPRESSION;
        OUTWORD(OPCODE+256*TAND1
                        +DISP(BTRNC));
     END;
    /* PULL,PUSH,XCHRS */
31: OUTWORD(EXPRESSION*256+OPCODE);
```

Figure 2 (continued)

```
                32: BEGIN    /* RADD,RAND,RCPY,RXCH,RXOR */
                         TAND1:=EXPRESSION;
                         TAND2:=EXPRESSION;
                        OUTWORD(OPCODE+256*(4*TAND1+TAND2));
                    END;
                33: OUTWORD(OPCODE);  /* HALT,PUSHF,PULLF */
                34: BEGIN       /* PFLG,SFLG */
                         TAND1:=EXPRESSION;
                         IF (TAND1>7) THEN ERROR(BADEX);
                         IF ¬EOLF
                         THEN TAND2:=ANDF(EXPRESSION,H007F)
                         ELSE TAND2:=0;
                         OUTWORD(OPCODE+256*TAND1+TAND2)
                    END

        END;
END;
```

Figure 2 (continued)

Criticism of the Program

In its present form, the MUMS assembler offers several interesting advantages in addition to those discussed previously. Concentually, the technique results in a collection of sibling assemblers which differ only in opcode mnemonics and in interpretation of operands. This uniformity between siblings guarantees that a user who is expert in the use of one sibling can learn to use another sibling by studying only the instruction set of the machine in question. The cost of training a programmer in the intricacies of a new AL is therefore minimized.

The current program also effectively packs five assemblers into the disk space required for one. However, the five languages implemented thus far require thirty-five templates. This makes for a rather large and unwieldy template procedure, and therefore suggests a practical limit on the number of languages that should be implemented by a single copy of the program. This proposed limitation is not for cosmetic purposes, since the appearance of the source listing has little relation to efficiency. Neither is core a problem, since the template procedure presently accounts for less than ten per cent of the total core requirements. The real objection is that an already poor execution time deteriorates further because of the search time required to select templates near the end of the case list. An elegant solution to this problem might entail dynamic selection and loading of a template procedure at run time in the same manner as opcode table initialization. This feature would further reduce development costs, since only a relatively short template procedure would have to be compiled and debugged. There is a remote possibility that this feature could be implemented with a recent (undocumented) version of the PASCAL compiler.

While PASCAL is a beautiful language from a maintenance standpoint, PASCAL programs usually do not execute very efficiently. For one thing, since PASCAL is procedure oriented, one faces the overwhelming temptation to make every subtask a procedure, even if only a single call may be

necessary. This produces a very readable source listing, but, unfortunately, it also creates unnecessary stack overhead at run time. In addition, the use of operators like AND, OR, and SHIFT seems natural for the task of object code construction. However, since DEC-10 PASCAL variables are strongly typed, "boolean" operators may not be applied to integer data types. This problem was rather inefficiently solved by writing functions ANDF and ORF, which perform the required boolean operations through use of the MACRO construct. This feature allows DEC-10 assembly language statements to be present in the PASCAL source. SHIFT was implemented by integer multiplication and division. However, when an arithmetic right shift with sign bit extended is required, one must be careful that the sign bits do not disappear (i.e., the result of the division is zero). This is insured by simply truncating unused high order bits prior to the shift.

Besides their gross inefficiency, the ANDF and ORF functions are clearly undesirable in terms of transportability. However, this is not viewed as a particularly serious defect, since DEC-10 programs are usually not transportable without a great deal of modification anyway. One major incompatibility stems from the lack of the standard PASCAL EOL character. Instead, DEC-10 PASCAL supplies two procedures, READLN, which sets EOLN to false and moves the "read head" past a CR-LF on the input medium, and WRITELN, which outputs a CR-LF. Many of

the built-in functions are oriented towards on-line use and require a file name in their parameter lists. In addition, the quality of the PASCAL compiler leaves much to be desired, as many things are implemented either poorly or not at all. For example, to read integer data types with the READ procedure, the read head must first be manually positioned to the start of the ASCII digit string via the GET procedure. Similarly, READLN does nothing if a CR-LF is not under the read head. Hence, the I/O routines in ASSEM will probably require considerable revision before they will operate properly elsewhere.

In order to gain some insight into performance, the MUMS assembler (ASSEM) was benchmarked against a local assembler (ASM16) for the IMP-16. ASM16 is actually a modified version of the DEC-10 MACRO assembler, and requires a minimum of 21P of core, where a page (P) is 512 thirty-six bit words [10]. In contrast, ASSEM occupies a minimum of 26P. However, these figures are slightly misleading. After compilation, a typical DEC-10 program resides on disk in two separate files, which are usually referred to as the low and high segments. The low segment contains information about constants, variables, and data structures, while the high segment contains absolute object code. The symbol table in ASM16 is managed by dynamic techniques, and hence starts with five pages of core. ASSEM uses fixed size tables, and must therefore request enough space to satisfy the largest number of symbols expected. The current version allows 500

symbols, thereby resulting in a low segment of 13P. Since the symbol table is hash coded, a large table decreases table search time since it maximizes the probability that a symbol will quickly hash into a unique location, albeit at the cost of sparse tables. During execution, ASSEM will require a few additional pages for stack management. The difference of 3P in the high segments is apparently due to the MACRO facilities in ASM16, which are, of course, not present in ASSEM.

The relative execution efficiencies of the two assemblers were compared by assembling similar IMP-16 programs. The test routine implemented the CORDIC rotation algorithm for generation of SIN and COS. Each version of this routine contained about 225 source lines and differed only in the pseudo-ops present. ASM16 handled the test data in 1.85 seconds, while ASSEM required 5.88 seconds. This rather poor showing is due primarily to the enormous overhead of character manipulation in ASSEM. All operations involving line scanning, symbol matching, and conversions from character string to internal integer values are performed on a per character basis. The lack of efficient string processing facilities in PASCAL means that the program has to do the work itself. DEC-10 PASCAL does allow string variables, but there are no functions available that do anything useful with them. Standard PASCAL provides only PACK and UNPACK, which are not yet implemented in the DEC-10 version. What are really needed in this application are

built-in functions similar to those provided by PL/C (e.g., the string concatenation operator, SUBSTR, INDEX, etc.). These functions could be simulated via the MACRO feature of DEC-10 PASCAL, but the resulting code would hardly lend itself to casual maintenance.

IV. APPLICATIONS

The current version of the MUMS assembler mimics five existing assembly languages. However, the same techniques can obviously be used to translate almost any AL-like source. This realization suggested the following applications.

Microprogramming

User microprogrammable minicomputers have been on the market for some time. The semiconductor houses also supply chip sets which allow the user to build his own microprogrammable hardware. Hence, there has been considerable interest in microlanguages. Vertical microinstructions perform single functions and typically have short word lengths. A corresponding microlanguage would then appear to be very similar to conventional assembly languages and could easily be implemented by the MUMS assembler. In constrast, horizontal microinstructions may simultaneously control several resources and therefore require much larger word lengths. Since horizontal microinstructions are very awkward to assemble by conventional AL techniques, the MUMS assembler is expected to have limited utility. An excellent introduction to these

problems may be found in [11].

Universal Assembly Language

The proliferation of inexpensive processors has greatly complicated the efforts of software designers. The question of "Which machine to use?" continues to consume many man-hours of investigation. However, even when the processor has been selected, many programmers complain about a problem analogous to re-inventing the wheel: before any serious software development can begin, the necessary development tools must be purchased or developed in-house. This development facility may reside solely on some host system, solely on the target machine itself, or some combination of the two. The components of this development facility are usually a subset of the following: an assembler, compiler, interpreter, and some basic software package for the target machine (e.g., a linking (?) loader, monitor, and a debug program). Since these components are obviously machine dependent, it may be prohibitively expensive to maintain development stations for more than one machine.

One solution of current interest is the use of a general purpose assembly language (GPAL). Conceptually, GPAL represents the instruction set of a carefully defined virtual machine. GPAL instructions have the "nice" property

that they map easily (not necessarily one-to-one) into instructions for any real machine. If all systems and applications programs were written in GPAL, this software could then be easily bootstrapped into a new machine by writing a simple translator. The design of GPAL is beyond the scope of this thesis. Hence, the following discussion will be limited to the use of the MUMS assembler with GPAL.

After completely specifying the GPAL machine, it is straightforward to implement an assembler for GPAL mnemonics. The GPAL object code may then be executed interpretively by any real machine. Address computation is performed by the interpreter, so that GPAL code is effectively position independent. Since only one assembler is involved, this method results in a clean interface with the host system. However, this technique is undesirable because a rather lengthy interpreter must be written in AL for each target machine. Since the interpreter must be resident, memory, still a rather valuable system resource, is wasted. Finally, the overhead of interpretation may degrade system performance intolerably.

Alternatively, each GPAL instruction could be assembled directly into the object code of the target machine. This technique corrects the above deficiencies, but it also lacks the advantages. However, any complexities in the host-target machine interface clearly belong on the host side. It is clearly preferable to write a set of templates

in HLL, as opposed to writing a set of interpreters in AL. Unfortunately, a unique template will probably be required for each GPAL instruction. This means that the template procedure will be fairly large, which dictates the use of one translator program per machine.

It is also possible to use the MUMS assembler as a translator from GPAL mnemonics to machine-X mnemonics. This feature would make it possible for the user to do his own optimization on GPAL routines after translation. As before, this function requires a large template procedure, and hence a dedicated program for each machine. Since GPAL is a logical choice for compiler pseudo-code, it is hoped that these ideas aid its development.

V. SUGGESTIONS FOR FUTURE WORK

The ideas in the preceding section are straightforward applications for the MUMS assembler. The following discussion will be concerned with options that require modifications to the skeleton. These options have so far been omitted because of lack of time, and because of a desire to keep the program as simple as possible. When it became apparent that the assembler was not particularly efficient, it also seemed logical to suppress the overhead associated with any unnecessary functions. However, future improvements in the skeleton should eliminate this objection. Finally, it should be remembered that the skeleton is really a collection of useful modules that can easily be incorporated into other PASCAL programs.

Conversational Assembler

Instead of using one of the system editors, some users may prefer to enter their programs via a conversational mode of the assembler. While the user enters his program, the assembler concurrently performs the first pass. If the current line has an obvious error, the assembler should complain and request that the line be retyped. Otherwise, the assembler will prompt for the next line. This feedback

increases the probability of an error-free first run. However, the user sacrifices much of the flexibility offered by the system text editors, and will probably spend more time debugging programs on-line, thereby increasing development costs. An effective compromise might involve taking the source program from disk as usual. During the first pass, the assembler could request that erroneous lines be re-entered. This is, of course, a selective error correction technique. For example, an unknown label in the operand field could be a valid forward reference, and should therefore be passed. Errors detected during pass two (e.g., undeclared label, or a branch address out of range) are much more serious, so that the user should be forced to make his repairs with the system editor. This built-in editing function offers some convenience, as it shortens the edit-assemble cycle.

Macro Assembler

Most advanced programmers agree that a macro facility should definitely be present in an assembler. Macros increase programmer effectiveness and enhance program readability. In addition, a macro facility might offer a simple means of implementing GPAL. GPAL macros could exist in a separate disk file for each machine. Assembly would then proceed in three passes, with the macro substitutions occurring during the first pass. However, since the MUMS

assembler is already a somewhat large and clumsy program, it might be worthwhile to implement the macroprocessor as a separate program in a language better suited to the purpose, such as SNOBOL. If desired, the two programs could be linked at run time to accomplish the desired results.

Relocatable Output

The MUMS assembler currently produces absolute object code. While this is adequate in many situations, the flexibility of the relocatable scheme may eventually be desired.

Conclusion

Computer programming is, and probably always will be, an art. Hence, most programmers find it difficult to determine when their products are ready for release, as it is possible only in exceptional instances to state conclusively that a program is optimal in some respect, or even that it is correct. However, after a reasonable amount of testing, it is usually possible to establish a qualitative level of confidence in the software. In the case of the MUMS assembler, programs for several different machines have been successfully assembled and executed. Furthermore, the programming technique appears to be easily

extensible to other machines. As discussed in previous sections, the implementation of the assembler skeleton is relatively inefficient. This is due partly to the goals of the project, and partly to the limitations imposed by the DEC-10 PASCAL compiler. Thus, it appears that skeletal improvements may be a continuing effort over the life of the program. However, as it stands, the MUMS assembler is considered to be a functionally complete and useful program.

REFERENCES

[1] V. S. Lamb, "All About Cross-Assemblers," Datamation, vol. 19, No. 7, pp. 77-80, July 1973.

[2] C. Popper, "SMAL - A Structured Macro-Assembly Language For A Microprocessor," COMPCON FALL 74, September 1974.

[3] A. Opler, "Is Assembly Language Programming Passe?" Data Processing Digest, vol. 14, pp. 1-15, October 1968.

[4] S. W. Conley, "Portable Cross-Assemblers in BASIC," Computer, vol. 10, No. 10, pp. 32-42, October 1975.

[5] D. Jones, and S. Lathrop, "An Assembler for the INTEL 8080 on the PDP-11," CS491 class project, University of Illinois, Urbana, Illinois, Spring, 1975.

[6] C. Georgiou, "A Generalized Assembler," Master's thesis, University of Illinois, Urbana, Illinois, January 1969.

[7] "PDP-8 Assembler," U. S. Government Research and Development Reports, vol. 68, p. 96, November 10, 1968.

[8] T. Wilcox, CS326 class notes, University of Illinois, Urbana, Illinois, Spring 1976.

[9] IMP-16 APPLICATIONS MANUAL, National Semiconductor Corporation, Santa Clara, California, June 1973.

[10] DECSYSTEM10 MONITOR CALLS MANUAL, Digital Equipment Corporation, Maynard, Massachusetts, May 1974.

[11] A. Agrawala, and T. Rauscher, "Micro-programming: Perspective and Status," IEEE Trans. on Computers, vol. C-23, no. 8, pp. 817-837, August 1974.

APPENDIX

Supplied Pseudo-ops

.ASCII 'string'
The ASCII characters in the string are packed (left-justified) into consecutive memory locations. If necessary, blanks are inserted to fill the rightmost bytes of the last word. The string may be delimited by any character not occurring in the string except for ';', '_', and '='.

.BYTE exp1, exp2, ...
The values of the expressions are truncated to eight bits, and then loaded into consecutive locations of memory.

.END exp
This directive specifies the end of the current pass. On pass two, the optional expression specifies the starting address for load-and-go execution.

.FORMAT (HEX or OCTAL)
The argument of .FORMAT changes the default listing mode for addresses and object code to hex or octal.

.LIST turns on the assembly listing.

.NLIST turns off the assembly listing.

.NPUNCH suppresses paper tape generation for the entire program.

.PAGE forces a new page if .LIST is in effect.

.RADIX exp
The default radix for evaluation of constants is set to the value of the expression.

.TABLE outputs an alphabetically sorted symbol table.

.WORD exp1, exp2, ...
Consecutive words of memory are set to exp1, exp2, etc.

Description of EBF Notation

EBF is an extended version of BNF (Backus-Naur Form). EBF and BNF are metalanguages, i.e., languages used for describing other languages. In Tables 1 and 2, the rules are stated in the form

'left hand side' ::= 'right hand side'

where '::=' means "may be composed of" [8]. Items appearing on the left hand side of '::=' are called non-terminal symbols, and are non-terminal in the sense that they never appear explicitly in the actual text. Rather, the non-terminal represents any phrase that can be constructed from the rules of its right part. The right part may consist of terminal symbols, non-terminals, and other metanotation. Terminal symbols may appear in the source, and are usually mnemonic for the tokens they denote. In Tables 1 and 2, the terminal ID is an example of a class symbol. Class symbols denote all of the possible tokens that may be constructed from the corresponding set of lexical productions. In Table 2, note that the class symbols serve as labels for the appropriate productions.

Although BNF and EBF are capable of describing the same phrases, the trees generated by each notation are different. For example, the BNF rules

<list> ::= <num> <cnum>

<cnum> ::= <empty> | COMMA <num> <cnum>

describe a list of numbers. However, <cnum> is right recursive, and therefore has a right-branching subtree. A hierarchical phrase structure is clearly undesirable for a list. Hence, the following EBF construction, which describes a 'bushy' subtree, is generally preferred.

<list> ::= (<num>++COMMA)

Other EBF notation is described in Table 5.

TABLE 5

Examples of EBF Notation

EBF notation	Phrases produced by notation
...(<a>|<b>)...	...<a>... or ...<b>...
...<a><b>...	concatenation of phrases <a> and <b>
...<a>**n...	...<a> ... <a>... |_____| n-times
...<a>+*n...	...<a>... | ...<a><a>... . . | ...<a> ... <a>... |_____| n-times
...<a>*...	...(zero or more occurrences of <a>)...
...<a>+...	...(one or more occurrences of <a>)...
...(<a>)?...	...(zero or one occurrence of <a>)...

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-76-803	2.	3. Recipient's Accession No.

4. Title and Subtitle

A UNIVERSAL CROSS-ASSEMBLER

5. Report Date: May 1976

6.

7. Author(s)
Charles Patrick Kominczak

8. Performing Organization Rept. No. UIUCDCS-R-76-803

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
Urbana, Illinois 61081

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

Department of Electrical Engineering
University of Illinois
Urbana, Illinois 61081

13. Type of Report & Period Covered
Master's Thesis

14.

15. Supplementary Notes

16. Abstracts

This report is not a tutorial on the art of writing assemblers. Rather, it is a discussion of those properties that all assemblers have in common, and of a technique by which new cross-assemblers may be implemented very quickly. This research was motivated by the current proliferation of microporcessors and the associated software "vacuum." The result is a single cross-assembler capable of assembling absolute object code for five different microprocessors.

17. Key Words and Document Analysis. 17a. Descriptors

Assemblers
Microprocessors

7b. Identifiers/Open-Ended Terms

7c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	51
	20. Security Class (This Page) UNCLASSIFIED	22. Price --

JUL 8 1975

JUN 2 1 REC'D

BOUND BY HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.800-805(1976
Internet report /
3 0112 088402786